Master Thesis
-
Using Automatic differentiation to find gradients for recurrent neural networks in Futhark

Andreas Nicolaisen
JTC303@alumni.ku.dk
Primary supervisor: Cosmin Eugen Oancea
Secondary supervisor: Stefan Oehmcke
Submission: December 20, 2021

December 20, 2021

1 Abstract

In this report, we will explain how Backwards automatic differentiation, can be used to transform an implementation of a recurrent neural network of the programming language, into a program that can find gradients for the weights that are given to the network. We will give the relevant background for neural networks, Futhark, and gradient descent as background. We will present different methods for finding gradients, discuss their pros and cons and explain why we chose the method of backwards automatic differentiation. We will explain how the main rewrite rule of backwards automatic differentiation,can be used to create new rules for constructs in Futhark, and then we will use those rule to transform an implementation of the Elman neural network, into a program that finds gradients for the input weights. Lastly, we will validate the results against an implementation in the machine learning framework PyTorch, and discuss future work.

2 Introduction

In this report, we will present an attempt at performing automatic differentiation, on recurrent neural networks (RNNs), with the purpose of finding gradient to train the network with. The implementation of the recurrent neural networks is implemented in the programming language Futhark, which is well suited for this sort of task. To do this, we will first present some relevant background information regarding "simpler" neural networks, the gradient descent method, as well as relevant parts of the Futhark language. We will then explain RNNs (how they work), and give 3 examples of RNNs. We will then discuss different ways of finding gradients, each with their advantages and disadvantages, and pick one that we will use. We will then explain how to use the method that we pick (Backwards automatic differentiation) in Futhark, and derive some rules that we will use in the implementation. Then we will give a brief overview of how the Elman RNN is implemented, for normal prediction purposes. Then we will break down the implementation using the rules we derived earlier, in order to create a version of it, from which we can get the gradients we need to train the model. Then we will validate the implementation, against a similar implementation in PyTorch, to make sure that we're getting the correct gradients. We will also give some run times for the implementation. Finally, we will conclude the project, and discuss possible future work.

3 Background

3.1 Neural networks

Here, we will give a very brief introduction to how a "normal" neural network works. We start with an input of type $\mathbb{R}^d$. We want to create a model which takes input of this shape, and makes some "prediction", in our case it we be an output also of type $\mathbb{R}^d$. In order to make this prediction, we first pass the values of the input to a layer of "neurons" (each neuron gets all of the input values), which each calculates a value based on a weight, the input, and some bias. This values is then passed to an "activation function", which returns a values which is taken to be the output of the neuron. For a single neuron with an output h, the calculation looks like this, $\cdot$ being vector dot product:

$$h = \sigma(w \cdot x + b) \tag{1}$$
$$h \in \mathbb{R} \tag{2}$$
$$b \in \mathbb{R} \tag{3}$$
$$x \in \mathbb{R}^d \tag{4}$$
$$w \in \mathbb{R}^d \tag{5}$$
$$\sigma \in \mathbb{R} \to \mathbb{R} \tag{6}$$

Once h for each neuron has been calculated, they are passed to the next layer in the network, as can been see in figure 1, which passes to the next layer, until it reaches the output. Using matrix-vector multiplication, we can more efficiently calculate all of the h's in a layer, but doing it at the same time. If we have multiple inputs, using matrix-matrix multiplication, we can calculate the h's for an entire layer, for all of the inputs at the same time. More on this later.

Activation functions

Activation functions are a set of functions, that are are used to determine how "active" a specific neuron should be, given some input to the activation function, which is calculated in the neuron. The result of the activation function is what is considered as the "output" of the neuron. A typical (but not given) property of an activation function, is that it "squashes" the input to it, into some constrained range, such as $(0, 1)$, $(-1, 1)$. A desirable property is that the activation function has a known, and well behaved derivative, since this makes it easier to find gradients when training the model (see later sections for what the gradient is, and why finding it matters). The activation function that we will be using is the $tanh$, also known as hyperbolic tangent,

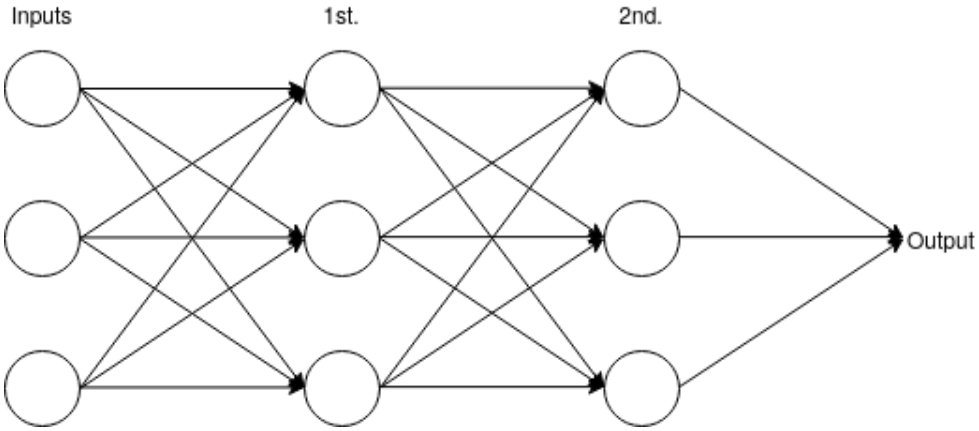

Figure 1: Diagram of a NN, with dimensionality of 3, and with 2 hidden layers. Each of the solid lines represents a scalar

function. There is no particular reason for why this one is chosen, other than it is a common one, and it has a well behaved, and kwown derivative. The formula for it is the following:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{7}$$

And its derivative is the following:

$$tanh'(x) = 1 - tanh(x)^2 \tag{8}$$

3.2 Dimensionalities of inputs and hidden layers

For real-world scenarios, we might have an input dimensionality of say 50 (these will be made up numbers, but should serve to illustrate a point), but want to have hidden layers with, say 100 in each. In order to accomplish this, the first of the 100-"width" would be set up to only be expecting 50 inputs to each neuron, and still produce their normal outputs, typically a single scalar each, resulting in 100 total values. Then any subsequent layers would be set up such that each neuron would expect 100 values, and produce a scalar values each. Similarly, we could also setup a layer with a reduced width, compared to the preceding layer. For simplicity, in this project, we will only be working with networks with the same dimensionality all the way through, as in, the input, hidden layer and output sizes will all be the same.

3.3 Futhark

This project will be implemented using the programming language Futhark. Futhark is a purely functional, array language supporting nested parallelism [7] by means of second-order array combinators (SOAC), such as map, reduce, scan, scatter, reduce-by-index [6], but also loops with in-place updates.

The constructs that we will be using in this project are map, reduce, (do) loops, and in place updated.

map is a build in function, which takes as input a function, and an array whose elements the function takes as an argument. It applies the function to each of the elements, and produces a new array, with the resulting values. Variants such as *map2* exists, which takes to a function that operators on to arguments, and two arrays whose element types correspond to those arguments.

reduce is also a build in function, which takes as arguments an binary operator, a "neutral element", and an array. The operator has to apply to the types of the elements in the array, and in order for the reduce to be parallelized, the operator has to be associative. The neutral element is an element, which has the property that when the operator is applied to any element, and the neutral element, the result is the original element. For the (+) operator (which is the only one we will be using with reduce), the neutral element is 0, since $a + 0 = a$.

We will also be using *do loops*, with in-place updates of arrays. In *do loops*, we define the loop-variant(s), which in this case will include the arrays that we want to do in-place updates to, as well as some sequence, whose elements we will loop over (for all cases here, this will be a sequence if integers). The loop works by binding an initial value to the loop-variant(s), and then in each iteration doing the operations that we want to, and return a new version of the loop-variant(s) at the end of the loop body. As mentioned, if we want to do in-place updates to an array (such as saving something from each iteration to an array), we need to include the array in the loop-variants. This allows the Futhark compiler to check for uniqueness.

Futhark as previously been used to speed up big-compute applications, such as detecting landscape changes from analysis of satellite time-series images applied at continental scale [4].

3.4 PyTorch

PyTorch is a machine learning framework which features automatic differentiation [9], and we will be using it for validating the implementation made in Futhark.

3.5 Gradient descent

In order to make neural networks, including recurrent ones, we need some way of find good parameters to use. To do this, we first need to have some notion of what makes a good parameters. This is done using some objective function, and a training set. The model is used on the training set, with some initial weights, and some predictions are made. The predictions are compared with some corresponding "true values", according to the objective function. A common one, and the one used here, is the "mean squared error", abbreviated "mse":

$$mse(predictions, truths) = \frac{1}{n}\sum_{i}^{n}(truths_i - prediction_i)^2 \tag{9}$$

This results in a value, telling us how good the current weights are for predicting the true values. From this, we would the like to find a new set of weights that would accomplish this better. To do this, we want to figure out how the different values in the current weights influence the final result. In other words, we want to differentiate the result with respect to the individual weights. This is know as finding the gradients, and there are multiple ways of doing it, some of which will be discussed in the next section. When we have found these gradient, we will subtract their values, usually multiplied with a "learning rate", from the original weights. This purpose of this is to minimize the value of the loss, or at least make it smaller then before. This process can then be repeated for a number of times, usually some predetermined number of times, or until the loss doesn't change significantly.

4 What are RNNs?

Recurrent neural networks are a type of neural networks that process an ordered "time-"series of inputs[1], instead of only single inputs. Examples of series could be a series of sensor readings, stock data at intervals for some time period, texts in natural language or speech.

[1] explain

RNNs works like traditional neural works but in addition to taking their normal inputs, the layers also use some internal state to calculate their outputs. In addition to their normal outputs, they also produce a new state, which is used when the next input in the time series is processed. Different weights might be used for controlling how much is taken from the input and the existing state, for calculating the output, as well as the new state, in addition to biases. For a visualization of how the a single input passes through a RNN, see figure 2.

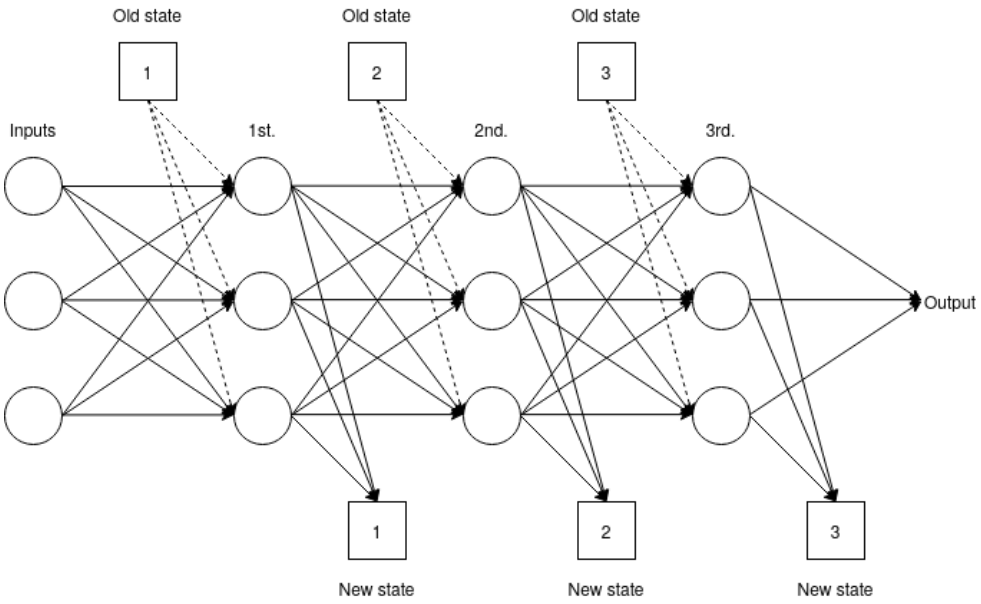

Figure 2: Diagram of a RNN, with dimensionality of 3, and with 3 layers. Each of the solid lines represents a scalar, and each of the stippled lines represents a vector of scalars. Note here that the 3 lines coming from each of the "Old state" boxes represents the same vector. The 3 solid lines going to each of the "New state" boxes, is used to create a new vector.

4.1 Elman

The Elman network, is a recurrent neural network, and it was first proposed by Jeffrey L. Elman[3]. For each neuron in the network, it uses 3 weights, 2 of them being a vector. The first one, typically called w_h is a vector of size d, d being the dimensionality of the input, the second weight being u_h also being a vector of size d, and the last one being a scalar value called b_h

signifying the bias. The formula for calculating the output is the following, $\cdot$ signifying vector dot product:

$$h = \sigma(w_h \cdot x + u_h \cdot last_h + b_h) \tag{10}$$
$$h \in \mathbb{R} \tag{11}$$
$$b_h \in \mathbb{R} \tag{12}$$
$$x \in \mathbb{R}^d \tag{13}$$
$$last_h \in \mathbb{R}^d \tag{14}$$
$$w_h \in \mathbb{R}^d \tag{15}$$
$$u_h \in \mathbb{R}^d \tag{16}$$
$$\sigma \in \mathbb{R} \to \mathbb{R} \tag{17}$$

Here, σ is the activation function, x is the input to the layer and $last_h$ is the existing state. The h that we compute here is then the new state, which is saved and then used in place of $last_h$ in the processing of the next input of the sequence. Some versions of the Elman network then goes on to add an additional bias to the h that we found, and run that through an additional activation function, in order to find the output of the layer. The version that we're considering uses the same h as the output of the layer.

Like normal NNs, we can efficiently calculate the outputs, as well as the new states, for the *entire layer* at the same time. For a layer the number of neurons being w, also known as the "width" of the layers, we can do this arranging the weights into a matrix of dimensionality $\mathbb{R}^{w,d}$. We can then perform matrix-vector multiplication between the arranged weights, and the input to the layer. We will have to do this for both $w_h \cdot x$ and $u_h \cdot last_h$. We will get 2 vectors from this, which we can add together with the biases arranged into a vector as well. The formula for this look like this, with $\times$ being matrix-vector multiplication, the $+$ being overloaded to perform vector-vector addition and the semantics of σ being the activation function

applied element-wise (essentially like a map in Futhark):

$$hs = \sigma(w_{hs} \times x + u_{hs} \times last_{hs} + b_{hs}) \tag{18}$$
$$hs \in \mathbb{R}^w \tag{19}$$
$$b_{hs} \in \mathbb{R}^w \tag{20}$$
$$x \in \mathbb{R}^d \tag{21}$$
$$last_{hs} \in \mathbb{R}^d \tag{22}$$
$$w_{hs} \in \mathbb{R}^{w,d} \tag{23}$$
$$u_{hs} \in \mathbb{R}^{w,w} \tag{24}$$
$$\sigma \in \mathbb{R}^w \to \mathbb{R}^w \tag{25}$$

Further, if we have batches of data (ie. multiple series), we can calculate the outputs and states for a layer, for the entire batch at a time. This is done by arranging the inputs into the columns of a matrix. Then we can perform matrix-matrix multiplication and addition, similarly to how we performed matrix-vector multiplication and addition before.

4.2 GRU

The "gated recurrent unit" is a recurrent neural network[2]. It is more complex than the Elman network and it features more weights. The formula for computing a the values in a layer is the following ($\odot$ for element-wise multiplication, $\cdot$ for matrix-vector multiplication):

$$r = \sigma_r(w_r \cdot x + u_r \cdot last_h + b_r) \tag{26}$$
$$z = \sigma_z(w_z \cdot x + u_z \cdot last_h + b_z) \tag{27}$$
$$n = \sigma_n(w_n \cdot x + r \odot (u_n \cdot last_h) + b_n) \tag{28}$$
$$h = z \odot (last_h) + (1 - z) \odot n \tag{29}$$
$$h \in \mathbb{R} \tag{30}$$
$$x \in \mathbb{R} \tag{31}$$
$$b_r, b_z, b_n \in \mathbb{R}^d \tag{32}$$
$$w_r, w_z, w_n \in \mathbb{R}^{w,d} \tag{33}$$
$$u_r, u_z, u_n \in \mathbb{R}^{w,w} \tag{34}$$
$$\sigma_r, \sigma_z, \sigma_n \in \mathbb{R}^w \to \mathbb{R}^w \tag{35}$$

Again, h is the both the output and the new state of the layer. $\sigma_r, \sigma_z, \sigma_n$ are all activation functions, similar to the activation function used in Elman. Similar to Elman, this can also be adapted to work with batch sets directly.

4.3 LSTM

We have also implemented the "Long short-term memory" RNN[8][5]

5 Finding gradients

As mentioned in the previous section, in order to train neural networks, we need to be able to find the gradients of a set of weights. There are multiple ways of finding these, and we will present and discuss some of them here.

5.1 Common methods

Numerical differentiation

The numerical method of finding gradients works by sampling the function in question at separate points. It is based on the limit definition of derivatives, and is relatively simple to implement. The formula for finding the derivative with respect to input variable x_i can be see in figure 3, with h being some small number that is decided by the implementer.

$$\frac{\partial f(x)}{\partial x_i} = \frac{f(x + he_i) - f(x)}{h} \tag{36}$$

Figure 3: Forward difference numerical differentiation of f, with respect to the ith input parameter

As mentioned, it is simple to implement, since the function in question will have to be implemented anyway, so using this to find derivatives is straight forward. However, using it comes with multiple challenges.

First of all, we need to decide the value of h. At first, it might seem like a good idea to pick small value for h. This is because the result will then more closely mirror the result of the limit definition for derivatives. This might work well, if the sizes of the input variables are small, and therefore good numerical precision is maintained, when using floating point numbers. If the sizes of the larger however, or we want to use floating point numbers with fewer bits, a small value of h will lead to imprecision. If we were to pick a lager value of h, we might maintain better numerical precision, or the same precision with fewer bits, but the approximation that we end up with will likely be worse. For an illustration of why this is, see figure 4

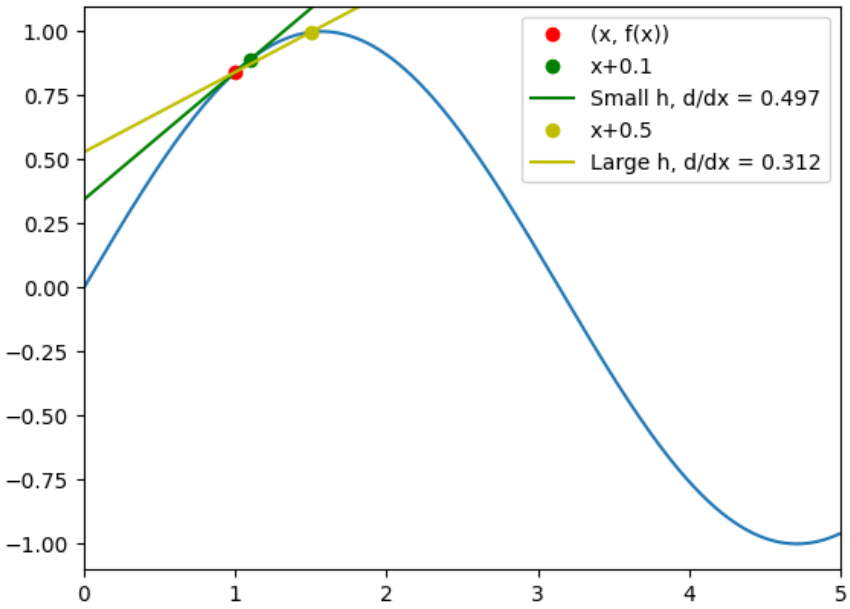

Figure 4: Illustration of the different outcomes, for different values of h. The true derivative value is ≈ 0.540

A second drawback of using numerical differentiation, is that we can only find the derivative value with respect to a single input parameter, at a time. If the number of input parameters is small, and the function itself quick to compute, this might be okay, but when there are many input parameters, and the function is complex, it will quickly become very expensive. Therefore, it is not well suited to use for computing the gradients needed when training a neural network, since it will most often have many tune-able parameters, and it is expensive to compute the result for each one.

Manual/Symbolic differentiation

Symbolic differentiation is in a sense the "traditional" way of differentiating functions. We start with the original function expression, and apply a series of known rules in order to obtain a new expression for the derivative of the function. This is essentially a automatic version of the differentiation that you can do by hand. While this methods gives exact derivatives, it comes with some limitations and inefficiencies.

In order to do symbolic differentiation, the functions has to be closed form. This might be fine for some models, but it excludes more complicated control flow like if-statements.

5.2 Automatic differentiation

The method of automatic differentiation is to perform differentiation based on how the expression is evaluation is performed, instead of the expression itself [1]. This is done by performing symbolic differentiation on the elementary operations of the evaluation, and computing gradients through intermediary values. There are 2 schemes for doing this, the forwards mode, and the backwards mode. They both come with advantages and disadvantages, and we will discuss them in this section. In order to explain them best, we will use the following function as a running example:

```
f(x_1, x_2) =
  a = 0
  b = 0
  for x < 2 do
    a += x_1 * x_2
    b += x_1 + x_2
  return (a, b)
```

Forward mode

In forward mode, we execute the function as we would normally, but between each elementary step, we determine the derivative of each intermediary value with respect to the input parameters. This derivatives can only be determined for one input parameter at a time. This is done by initializing a set of "derivative input parameters", all being zero, except the one we're currently seeking to find the derivative of, which is set to one. At each step in the execution, we find the derivative of the intermediary value, using the existing derivatives of other intermediary values, or the derivatives of the input parameters. See table 5

Filling out the Jacobian matrix for the example, we get (with question marks for the values that we don't know):

$$\begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 8 & ? \\ 2 & ? \end{bmatrix} \tag{37}$$

Figure 5: Forward mode automatic differentiation, with respect to input x_1

Program line	Forward trace		Forward derivative trace	
	$v_{-1} = x_1$	$= 3$	$\dot{v}_{-1} = \dot{x}_1$	$= 1$
	$v_0 = x_2$	$= 4$	$\dot{v}_{-0} = \dot{x}_2$	$= 0$
`a = 0`	$v_1 = 0$	$= 0$	$\dot{v}_1 = 0$	$= 0$
`b = 0`	$v_2 = 0$	$= 0$	$\dot{v}_2 = 0$	$= 0$
`x_1 * x_2`	$v_3 = v_{-1} \cdot v_0$	$= 12$	$\dot{v}_3 = \dot{v}_1 \cdot v_0 + \dot{v}_0 \cdot v_0$	$= 4$
`a += (x_1 * x_2)`	$v_4 = v_1 + v_3$	$= 12$	$\dot{v}_4 = \dot{v}_1 + \dot{v}_3$	$= 4$
`x_1 + x_2`	$v_5 = v_{-1} + v_0$	$= 7$	$\dot{v}_5 = \dot{v}_{-1} + \dot{v}_0$	$= 1$
`b += (x_1 + x_2)`	$v_6 = v_2 + v_5$	$= 7$	$\dot{v}_6 = \dot{v}_2 + \dot{v}_5$	$= 1$
`x_1 + x_2`	$v_7 = v_{-1} + v_0$	$= 12$	$\dot{v}_7 = \dot{v}_1 \cdot v_0 + \dot{v}_0 \cdot v_0$	$= 4$
`a += (x_1 * x_2)`	$v_8 = v_4 + v_7$	$= 24$	$\dot{v}_8 = \dot{v}_4 + \dot{v}_7$	$= 8$
`x_1 + x_2`	$v_9 = v_{-1} + v_0$	$= 7$	$\dot{v}_9 = \dot{v}_{-1} + \dot{v}_0$	$= 1$
`b += (x_1 + x_2)`	$v_{10} = v_6 + v_9$	$= 14$	$\dot{v}_{10} = \dot{v}_6 + \dot{v}_9$	$= 2$
`return a`	$y_1 = v_8$	$= 24$	$\dot{y}_1 = \dot{v}_8$	$= 8$
`return b`	$y_2 = v_{10}$	$= 14$	$\dot{y}_2 = \dot{v}_{10}$	$= 2$

As we can see, the forwards mode AD, fills out the columns in the Jacobian matrix. This means that in order to fill it out entirely, we need to run the forwards mode as many times as we have input variables. For a small amount of input variables, with a large amount of outputs, this seems like an efficient method, but we will compare the two in a later section.

Formulating the forwards mode AD as a rewrite for a Futhark program, we get the following:

$$\textbf{let } v = f(a,b) \Longrightarrow \textbf{let } v = f(a,b) \tag{38}$$

$$\textbf{let } \dot{v} = \frac{\partial f(a,b)}{\partial a}\dot{a} + \frac{\partial f(a,b)}{\partial b}\dot{b} \tag{39}$$

Backwards mode

In backwards mode automatic differentiation, we again execute the program as normally, while (theoretically) saving all of the intermediary variables. Then, we "assign" of one to one of the output variables. We then perform a backwards trace, were we for each elementary operation calculate the derivatives of any variables going into it, accumulation if existing derivatives already exist. Take for example the following 2 operations, and return:

```
f(v_0, v_1) =
  v_2 = 5 * v_1
  v_3 = v_2 * v_0
```

```
return v_2, v_3
```

We start with $v_0 = 4$ and $v_1 = 3$. Executing the code, we get the result of $15, 60$. Let's say we're interested in finding the derivatives of the input variables, wrt. v_3. To do this we start by assigning $\bar{v}_2 = 0, \bar{v}_3 = 1$, $\bar{v}_x$ being the derivative of v_x. We then look at the first operation, going backwards, namely $v_3 = v_2 \cdot v_0$, and find the derivatives wrt. to the 2 different input variables:

$$\frac{\partial v_3}{\partial v_2} v_2 \cdot v_0 = v_0$$
$$\frac{\partial v_3}{\partial v_0} v_2 \cdot v_0 = v_2$$

We then multiply these derivatives with the derivative of the variable they're going into, and we get the derivatives of the variables wrt. the chosen derivative for the output. Additionally, we add the existing derivative of v_2 when calculating the new one. In this case it is zero, so it doesn't make a difference.

$$\bar{v}_2 = \bar{v}_3 \frac{\partial v_3}{\partial v_2} v_2 \cdot v_0 = 1 \cdot v_0 = 4$$
$$\bar{v}_0 = \bar{v}_2 + \bar{v}_3 \frac{\partial v_3}{\partial v_0} v_2 \cdot v_0 = 0 + 1 \cdot v_2 = 15$$

This completes finding the derivatives for the first operation (in the backwards direction). We then repeat the same procedure for the next operation, first finding the expressions for the derivatives, in this case there only being one:

$$\frac{\partial v_2}{\partial v_1} 5 \cdot v_1 = 5$$

We again multiply with the derivative of the "output" variable:

$$\bar{v}_1 = \bar{v}_2 \frac{\partial v_2}{\partial v_1} 5 \cdot v_1 = \bar{v}_2 \cdot 5 = 20$$

This completes the backwards derivative trace of the program, and we end up with the final values of $\bar{v}_0 = \frac{v_3}{\partial v_0} = 15$ and $\bar{v}_1 = \frac{v_3}{\partial v_1} = 20$. We can put this into the Jacobian matrix, with question marks for the values that we didn't calculate here.:

$$J = \begin{bmatrix} \frac{\partial v_2}{\partial v_0} & \frac{\partial v_2}{\partial v_1} \\ \frac{\partial v_3}{\partial v_0} & \frac{\partial v_3}{\partial v_1} \end{bmatrix} = \begin{bmatrix} ? & ? \\ 15 & 20 \end{bmatrix} \tag{40}$$

If we then wanted to find the remaining derivatives, we would execute the backwards trace again, with $\bar{v}_2 = 1$ and $\bar{v}_3 = 0$. This would then fill out the other row, and we would have a complete Jacobian.

Figure 6: Backwards mode automatic differentiation, with respect to output a

Program line	Forward trace		Backwards derivative trace	
	$v_{-1} = x_1$	$= 3$	$\bar{x}_1 = \bar{v}_{-1}$	$= 8$
	$v_0 = x_2$	$= 4$	$\bar{x}_2 = \bar{v}_0$	$= 6$
`a = 0`	$v_1 = 0$	$= 0$		
`b = 0`	$v_2 = 0$	$= 0$		
			$\bar{v}_0 = \bar{v}_0 + \bar{v}_3 \frac{\partial v_3}{\partial v_0} = \bar{v}_0 + \bar{v}_3 \cdot v_{-1}$	$= 6$
`x_1 * x_2`	$v_3 = v_{-1} \cdot v_0$	$= 12$	$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_3 \frac{\partial v_3}{\partial v_1} = \bar{v}_{-1} + \bar{v}3 \cdot v_0$	$= 8$
			$\bar{v}_3 = \bar{v}_4 \frac{\partial v_4}{\partial v_3} = \bar{v}_4 \cdot 1$	$= 1$
`a += (x_1 * x_2)`	$v_4 = v_1 + v_3$	$= 12$	$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1} = \bar{v}_4 \cdot 1$	$= 1$
			$\bar{v}_0 = \bar{v}_0 + \bar{v}_5 \frac{\partial v_5}{\partial v_0} = \bar{v}_0 + \bar{v}5 \cdot 1$	$= 3$
`x_1 + x_2`	$v_5 = v_{-1} + v_0$	$= 7$	$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_5 \frac{\partial v_5}{\partial v_{-1}} = \bar{v}_{-1} + \bar{v}5 \cdot 1$	$= 4$
			$\bar{v}_5 = \bar{v}_6 \frac{\partial v_6}{\partial v_5} = \bar{v}_6 \cdot 1$	$= 0$
`b += (x_1 + x_2)`	$v_6 = v_2 + v_5$	$= 7$	$\bar{v}_2 = \bar{v}_6 \frac{\partial v_6}{\partial v_2} = \bar{v}_6 \cdot 1$	$= 0$
			$\bar{v}_0 = \bar{v}_0 + \bar{v}_7 \frac{\partial v_7}{\partial v_0} = 0 + \bar{v}_7 \cdot v_1$	$= 3$
`x_1 + x_2`	$v_7 = v_{-1} + v_0$	$= 12$	$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_7 \frac{\partial v_7}{\partial v_{-1}} = 0 + \bar{v}_7 \cdot v_0$	$= 4$
			$\bar{v}_7 = \bar{v}_8 \frac{\partial v_8}{\partial v_7} = \bar{v}_8 \cdot 1$	$= 1$
`a += (x_1 * x_2)`	$v_8 = v_4 + v_7$	$= 24$	$\bar{v}_4 = \bar{v}_8 \frac{\partial v_8}{\partial v_4} = \bar{v}_8 \cdot 1$	$= 1$
			$\bar{v}_0 = \bar{v}_9 \frac{\partial v_9}{\partial v_0} = \bar{v}_9 \cdot 1$	$= 0$
`x_1 + x_2`	$v_9 = v_{-1} + v_0$	$= 7$	$\bar{v}_{-1} = \bar{v}_9 \frac{\partial v_9}{\partial v_{-1}} = \bar{v}_9 \cdot 1$	$= 0$
			$\bar{v}_9 = \bar{v}_{10} \frac{\partial v_{10}}{\partial v_9} = \bar{v}_{10} \cdot 1$	$= 0$
`b += (x_1 + x_2)`	$v_{10} = v_6 + v_9$	$= 14$	$\bar{v}_6 = \bar{v}_{10} \frac{\partial v_{10}}{\partial v_6} = \bar{v}_{10} \cdot 1$	$= 0$
`return a`	$y_1 = v_8$	$= 24$	$\bar{v}_9$	1
`return b`	$y_2 = v_{10}$	$= 14$	$\bar{v}_{10}$	0

Formulating the backwards mode AD as a rewrite for a Futhark program, we get the following:

$$\textbf{let } v = f(a,b) \Longrightarrow \textbf{let } v = f(a,b) \tag{41}$$

$$\textbf{let } \bar{a} += \frac{\partial f(a,b)}{\partial a} \bar{v} \tag{42}$$

$$\textbf{let } \bar{b} += \frac{\partial f(a,b)}{\partial b} \bar{v} \tag{43}$$

Strengths and weaknesses

Overall, automatic differentiation has the advantage that it doesn't require us to find complicated derivative expressions. Instead we only have to find derivatives for simple expressions (only partial derivatives in the case of backwards AD), for which simple rules exist. This allows us to mechanically transform an existing program into a program which find the derivatives. The 2 different modes have different pros and cons with respect to each other, and have areas in which the excel compared to the other. In the design section we will discuss which is better for the purpose of finding gradients for neural networks.

Automatic differentiation comes with some relevant downsides as well. For forwards mode, we either have to store a lot of intermediary variables, or complicate the forwards trace with computing the derivatives at the same time. This will either take up memory, or be complicated to do, compared to if a (relatively) simple expression for the derivative could be find. This is even more the case for backwards AD, where we need to store the intermediary variables (or recompute them, see later sections). For both forwards and backwards, if there are multiple inputs or outputs respectively, we need to run the traces multiple times, which will of course take extra time. In general, if a simple expression for the derivative could be found, it would be more efficient. This however might be difficult or impossible to find.

6 Design considerations

6.1 Futhark

As mentioned earlier, Futhark is a language built to enable high performance, for big-compute applications. RNNs are an example of such, consisting mainly of matrix and vector operations, for which the sizes has the potential to be very large. It also comes with a lot of build-in functions, which are useful for implementing RNNs in a concise manner.

6.2 Forwards or backwards mode

When training neural networks, we need to find the gradients for a large amount of input variables (the weights), with respect to a single output, the loss. This makes the decision on which automatic differentiation mode to use quite clear, namely backwards mode, but for the sake of clarity, I will briefly explain the pros and cons of the different modes in the field of finding

gradients for neural networks.

As we discussed in the section about forwards mode AD, we can use it to find the derivatives of one variable, with respect to all of the output variables. This would be great if we had a small amount of input variables, and a large amount of output variables. But for RNNs, this couldn't be much further from the case. For RNNs, we have a large amount of input variables, the weights, and only a single output variable, the loss. This would mean that in order to find the gradient using forwards mode, we would need to execute the function (or at least the forward trace if all intermediary values are kept) as many times as we have weights. Hopefully, it should be clear that this would be a very inefficient technique for finding the gradients. We would be filling out the Jacobian matrix one column at a time, but given how there are a lot of columns (the number of weights), this is not the best method.

Considering backwards mode, we can quickly see why this has some clear advantages. Since we only have one output variable, we can compute all of the gradients that we need in a single backwards trace. This is because backwards mode fill out one row of the Jacobian matrix at a time, and we only have one row, since we only have one output. We do however, need to store a large amount of intermediary values to do it, or as we discuss in the next section, recompute them. Because of the clear advantage of only having to do one trace, this method of AD is the one that i have chosen.

6.3 Backwards AD in Futhark

From the previous sections, it should hopefully be clear that we could mechanically apply the rewrite rule for AD, and we would get a new program that could calculate the derivatives that we are looking for. For programs that consist of a straightline of scalar code (i.e., think a simple basic block), the application of the rewrite rule is straightforward. However, the extension of that (simple) rewrite rule to language constructs such as loops, map, reduce, or non-trivial functions is fairly non trivial.

This section investigates and presents the reasoning behind translating such (language) constructs for the reverse-AD mode, namely loops having statically-known counts (i.e., do loops), reduce and map.

Loops

The first construct that we examine is the (do) loop. The implementation of for example, the Elman network involves two perfectly nested loops, so it

should be clear that such a rule will be useful.

Let's first present of how a loop looks in Futhark:

```
1 let loopFunction [m] (n: i64) (x_inital : [m]f32) : [m]f32 =
2   let y =
3     loop (x) = (x_initial)
4       for i = 0 ... n-1 do
5         let res = body(x)
6         in res
7   in y
```

Here, the loop variant is x, which is initialized to *x_initial*. After each interation, the result of the body (*res*) is bound to x, i.e., it provides the value of the loop-variant array x that is input to the next iteration. The result returned in the final iteration is bound to y, which also become the result of the enclosing function.

If we wanted to perform backwards AD on the loop, we would run the code as we would normally, and record every operation that takes place in its, including the values being computed—this is known as "the tape". This would both be complicated to do, and would likely take up a lot of memory if n is large, and if there are a lot of intermediary variables in the body. Instead of saving everything, we could instead just save the values of x and the beginning of each iteration, and then recompute the values inside the body, during the backwards trace. This way, we get to perform backwards AD, without having to rely on a tape abstraction that remembers the result of the performed operations. Instead, we "checkpoint" the values of the loop-variant variables at the entry of each iteration, and the key idea is that we can always re-execute each iteration's body to bring to scope all the other intermediate variables. Essentially, the forward trace of a loop, checkpoints the loop-variant values of the loop, as demonstrated below:

```
1 let loop_AD_primal [m] (n: i64) (x_inital : [m]f32) : [m]f32 =
2   let xs_initial = replicate n (replicate m 0)
3   let (x_final, y) =
4     loop (xs, x) = (xs_initial, x_initial)
5     for i = 0 ... n-1 do
6       let xs[i] = x
7       let res = body(x)
8       in (xs, res)
9   in (x_final, y)
```

The statement that defines xs_initial creates a two-dimensional array—

by means of two nested replicate expressions—meaning, for each of the n iterations we will save its input, which is an array of length m of single-float values. This is accomplished in the first statement of the loop let xs[i] = x. Otherwise, the forward trace of a loop consists of the same code as the original loop. After executing the forward trace of the program we make use of the checkpointed information when we finally reach the the backwards trace of the loop, which is demonstrated below:

```
1   let loop_AD_Backwards n
2                         xs_checkpoint
3                         y_adjoint body_free_variables_adjoints =
4
5     let (new_y_adjoint, body_free_variables_adjoints) =
6       loop (y_adjoint', body_free_variables_adjoints') =
7         (y_adjoint, body_free_variables_adjoints)
8       for i = n-1 ... 0 do
9         let x = xs_checkpoint[i]
10        let res_adjoint = body_adjoint(x, y_adjoint',
11                                       body_free_variables_adjoints')
12        in (res_adjoint)
13    in
```

Here, quite a bit is happening. Let's first explain the changed arguments. We still have n, which is as before, but now we're also taking *xs_checkpoint* which is the *x_final* that was returned in the primal function. *y_adjoint* and *body_free_variables_adjoints* are the existing adjoint values of y and the free variables that could exist in the body function, as we enter into this piece of the larger program. Notice that the loop is now running backwards, since we're now doing the backwards trace. We can see that at the beginning of each iteration, we recover the state that existed, going into that iteration in from the check pointed values. We then pass this state in to the function *body_adjoint*, whose responsibility is then to perform the primal and backwards traces, for the body itself. Just to make it clear, a transformation of the body function has also happened in order to make it into *body_adjoint*, but what this transformation will be, depends on the content of the function.

Reduce

We could make a general rule for the *reduce* statement, that would work no matter which operator is being used with it, but in all of the instances that we're using it in, it is with the plus operator. Let's therefore focus on this special case. First, let's look at a reduce statement with plus as the operator:

```
1   let y = reduce (+) 0 xs
```

Here, y is the result of the reduce, 0 is the neutral element for the the operator *(+)* and xs is the arrary that is being summed together. Consider an "expanded" formulation of this:

$$y = left_i + xs_i + right_i \tag{44}$$

Here, xs_i is the ith element in xs, $left_i$ is the sum of all of the elements that came before xs_i, and $right_i$ is the sum of all of the elements that comes after xs_i. Let's consider what the adjoint of xs_i should be, using the backwards AD method:

$$\bar{xs_i} += \bar{y}\frac{\partial(left_i + xs_i + right_i)}{\partial xs_i} \tag{45}$$

Since $\frac{\partial(left_i+xs_i+right_i)}{\partial xs_i}$ is simply 1, we get that

$$\bar{xs_i} += \bar{y} \tag{46}$$

Here, $\bar{y}$ refers to the adjoint of the result. This of course applies for all of the elements of xs

Map

Finally, we're also using multiple instances of the map statement. There is one where the function given to the map has no free variables. For this case, since the statements in the functions can't affect the adjoints of any other variables, than the input variables, the backwards trace is essentially just finding the adjoint of the body itself, which is then multiplied that with the existing adjoint for the that entry in the array that is being mapped over.

6.4 Matrix-matrix multiplication

We also have another map in the program, which does have free variables in the body, namely matrix-matrix multiplication (and the special case of matrix-vector multiplication). In stead of trying to come up with a rewrite rule via the map function, let's look at an imperative version of matrix-matrix multiplication instead, between 2 matrices $A \in \mathbb{R}^{m,g}$ and $B \in \mathbb{R}^{g,n}$, into a new matrix $C \in \mathbb{R}^{g,n}$:

```
forall i = 0 ... m-1 do
  forall j = 0 ... n-1 do
    c[i,j] = 0
    forall k = 0 .. g-1 do
      c[i,j] +=A[i,k] * B[k,j]
```

Applying the original rewrite rule on the inner most statement we get, assuming we already know the adjoint of the result, in the form of `c_bar`:

```
forall i = 0 ... m-1 do
  forall j = 0 ... n-1 do
    forall k = 0 .. g-1 do
      a_bar[i,k] += b[k,j] * c_bar[i,j]
      b_bar[k,j] += a[i,k] * c_bar[i,j]
```

This gives us the adjoints of the 2 matrices that we were multiplying, which is what we sought. This imperative code, and then be transformed into something that we can implement in Futhark.

6.5 Checkpointing and redundant computation

As we saw in section about backwards AD, we need to know the values of the intermediary variables, in order to find th gradients. A simple way to accomplish this, would be to simply save all of the intermediary values during the forward pass, and then use them during the backwards pass. For an RNN, with many layers, and long series however, this could become a huge amount of values that need to be saved. A solution to this issue could be to redundantly compute the values of the intermediary variables, as we need them.

Let's consider how backwards AD works for an RNN. The very first thing we need to look at (with the exception of the computations regarding the losses) is the final layer in the network. To compute the gradients, we would then need the intermediary values used to compute the output of the final layer. To redundantly compute these, we will need the output values of the previous layer, which is computed in the layer previous to it, all the way back to the first layer; Secondly, and more critically, we also need the existing states for the this and the previous layers, and in order to compute that, we need to first have ran the RNN for all of the previous inputs, essentially just executing the entire function again. Let's say we did that, and found the gradients of the very final layer. We then need to go to the second to last layer, and essentially execute the entire function again. We could save some of the recomputed values, so we would have to re-execute at each step, but no matter what, we would still have to do a significant amount of recomputation. The reason it would not make sense to save a very significant amount of values during redundant computation, is because at that point, we might as well just save them during the forwards pass. An idea for how to reach a compromise between redundant computation, and not using too

much memory (and possibly slowing down because of it), is to introduce the concept of *checkpointing*.

The idea of checkpointing, is to save *some* values during the forward pass, while recomputing others during the backwards pass. An idea for which variables prioritize saving, could be those that take many operations to recompute. For a general example, this could be true loop variables, especially of outer loops. Specifically for RNNs, this would be the existing states that are used in each iteration of the outer most loops. For Elman networks, this is especially useful, since the state that is returned from a layer, is the same as the output of that layer. We then only need to recompute the intermediary variables involved with calculating the output of a layer in the first place. These would take up a lot of memory to store (5 vectors for single series/-matrices for batch mode), but a relatively simple to recompute if the state and input variable are known.

The primary scheme that I have picked is to save the resulting states after each input in the series have been processed. This approach will be compared with a version, where the intermediary variables for computing the output in each layer was saved, and a version where nothing was saved.

7 Implementation

I have implemented the Elman neural network, in both the forwards pass, and the backwards trance.

7.1 Forwards trace

I have implemented 2 versions of the Elman neural network, one that handles a single time series, and one that handles batches of time series. The implementations follows the definition relatively straight forward.

In the following, we can see the implementation of a layer, for the version that takes a single input (ie. not the batch version).

```
1 let elman_layer [d]
2           (x: [d]real) (last_h: [d]real)
3           (wh: [d][d]real) (u: [d][d]real) (bh: [d]real)
4           : ([d]real) =
5   let wx  = matvec_mul wh x
6   let ulh = matvec_mul u last_h
```

```
7     let h    = map (recurrent_layer_activation)
8                    (vector_add (vector_add wx ulh) bh)
9     in h
```

As we can see, the bulk of the work consist of performing matrix-vector multiplication, and then finally adding the vectors together, and running the activation function on each of the values in the vector. The activation function that i have chosen is the hyperbolic tangent, also known as "tanh" which is defined like this:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{47}$$

This has been implemented pretty straight forward:

```
1  let recurrent_layer_activation x =
2    (exp x - exp (-1.0 * x)) / (exp x + exp (-1.0 * x))
```

Since the vector-vector addition and matrix-vector multiplication will be explained in the next section, I will omit them here. The *elman_layer* function is called from *elman_predict* function:

```
1  let elman_predict [d][l]
2            (input: [d]real) (last_hs: [l][d]real)
3            (wh: [l][d][d]real) (u: [l][d][d]real) (bh: [l][d]real)
4            : ([l][d]real) =
5    let state_ini = replicate l <| replicate d 0
6    let (state', _) =
7      loop (s, x) = (state_ini, input)
8      for i < l do
9        let h = elman_layer x last_hs[i] wh[i] u[i] bh[i]
10       let s[i] = h
11       in  (s, h)
12   in state'
```

Here, we're going though all of the layers, one after the other. Notice, that in each iteration we're saving h into an array, as well as passing it directly to the next layer. This is because h acts as both the new state for that layer, and as the output of that layer. Since we're saving all the states/outputs into the array, we're just returning that directly.

Finally, we have the outer most loop:

```
1  let elman_rnn [n][d][l]
2            (inputs: [n][d]real) (first_hs: [l][d]real)
3            (wh: [l][d][d]real) (u: [l][d][d]real) (bh: [l][d]real)
4            : [d]real =
5    let final_hs' =
```

```
6       loop hs = first_hs
7       for i < n do
8         let h' = elman_predict inputs[i] hs wh u bh
9         in  h'
10
11    let prediction = final_hs'[l-1]
12    in prediction
```

This is essentially the same as the inner loop, but instead of iterating through the layers, we're iterating through the elements in the time-series. In this case, we're considering the output of the final layer, for the final input, to be the prediction. This is just one version, it could also be valid to consider the output of the final layer, for each of the inputs to be the result, but that is no the version we have chosen here.

7.2 Backwards trace

In this section, I will go through how we construct the forwards and backwards traces on the implememtation of the Elman network. To make it a bit simpler, I have chosen to explain it for the non-batch version. The batch version is mostly the same, the main differences being that is uses matrix-matrix operations, instead of matrix-vector operations.

Outer loop

In the outer most loop of the Elman implementation, we're iterating over the elements in the time-series:

```
1  let elman_rnn [n][d][l]
2            (inputs: [n][d]real) (first_hs: [l][d]real)
3            (wh: [l][d][d]real) (u: [l][d][d]real) (bh: [l][d]real)
4            : [d]real =
5    let final_hs' =
6      loop hs = first_hs
7      for i < n do
8        let h' = elman_predict inputs[i] hs wh u bh
9        in  h'
10
11   let prediction = final_hs'[l-1]
12   in prediction
```

Here, the loop variant is h, which is $\mathbb{R}^{l,d}$. Using the rule from earlier, when constructing the forwards trace, we would normally save h in the beginning of each loop iteration. I have however chosen to save h for the first iteration

to the checkpoints array, before the loop begins, and then save at the end of each loop iteration. This is simplely to pack the checkpoints, and the result into the same array, and is merely for convinience. Here is the forwards trace, with the checkpointing:

```
1  let elman_rnn_checkpoints [n][d][l]
2            (inputs: [n][d]real) (first_hs: [l][d]real)
3            (wh: [l][d][d]real) (u: [l][d][d]real) (bh: [l][d]real)
4            : [][l][d]real =
5    let checkpoints_0s = replicate (n+1) <| replicate l <| replicate d 0.0
6    let checkpoints_0s[0] = first_hs
7
8    let (final_checkpoints, _) =
9      loop (checkpoints, hs) = (checkpoints_0s, first_hs)
10     for i < n do
11       let h' = elman_predict inputs[i] hs wh u bh
12       let checkpoints[i+1] = h'
13       in (checkpoints, h')
14
15   in final_checkpoints
```

The backwards trace is then a bit more complicated, here without the initialization of adjoint values to zero:

```
1   let (final_wh_adj, final_u_adj, final_bh_adj,
2        final_hs_adj, final_input_adj) =
3     loop (wh_adj, u_adj, bh_adj,
4           hs_adj, input_adj) =
5       (initial_wh_adj, initial_u_adj, initial_bh_adj,
6        initial_hs_adj, initial_inputs_adj)
7     for j < n do
8       let i = (n-1) - j
9       -- Start by getting state into scope
10      let input      = inputs[i]
11      let hs         = checkpoints[i]
12      let hs_check   = checkpoints[i+1]
13
14      -- Find adjoints of body
15      let (wh_adj', u_adj', bh_adj', hs_adj', input_adj') =
16        elman_predict_adjoint
17          -- Original input
18          input hs wh u bh
19          -- Checkpoints
20          hs_check
21          -- Original_adjoints
22          hs_adj
23
24      -- Add to existing adjoints
25      let wh_adj_' = map2 mat_add wh_adj wh_adj'
26      let u_adj_'  = map2 mat_add u_adj  u_adj'
27      let bh_adj_' = mat_add bh_adj bh_adj'
28      let hs_adj_' = hs_adj' -- Since state is reset every interation
29      let input_adj[i] = input_adj'
30      in  (wh_adj_', u_adj_', bh_adj_', hs_adj_', input_adj)
31
32  in (final_wh_adj, final_u_adj, final_bh_adj,
33      final_hs_adj, final_input_adj)
```

In the beginning of the loop (notice that we're not going backwards), we start by recovering the value of the loop variant in from *hs = checkpoints[i]*. Here, we also get *hs_check = checkpoints[i+1]*. This is not the input to the current iteration, but is still given to the adjoint function for the body. Essentially, this is because it will acts as checkpoints for a loop inside the body, but we will elaborate on this later. The *elman_predict_adjoint* function then returns the adjoint values it calculates. We add them to the existing adjoint values for the weights, since they (the values of them weights themselves) stay the same, we reset *hs_adj* (it could be omitted, since it is not used,

because it is the loop variant), and then finally save the calculated adjoint of the current input. This last part is returned, but since we have no control over the values of the inputs, it is not used for anything, but included here for completeness. We could have given the existing adjoints to the *elman_predict_adjoint* function, and accumulated them there, but in order to keep the amount of arguments from becoming large, we accumulate them outside. This does lead to some redundent initialization of adjoints, which might affect performance. This is the only place where we accumulate afterwards like this, in other places, the accumulation either takes place inside the afjoint function, or no accumulation is necessary (ie. it is only assigned once).

Inner loop

The forwards trace of *elman_predict* is just the function itself, since we can effectivily perform the checkpointing of it in *elman_rnn_checkpoints*, and the backwards trace is quite similar to that of *elman_rnn*, and I have thus chosen to omit it.

Layer function

Finally, we have *elman_layer* which forms the body of loop of *elman_predict*. It looks like this:

```
1 let elman_layer [d]
2           (x: [d]real) (last_h: [d]real)
3           (wh: [d][d]real) (u: [d][d]real) (bh: [d]real)
4           : ([d]real) =
5   let wx  = matvec_mul wh x
6   let ulh = matvec_mul u last_h
7   let h   = map (recurrent_layer_activation)
8                 (vector_add (vector_add wx ulh) bh)
9   in h
```

Looking at this, we can see that we have a map statement, with the function being *recurrent_layer_activation*, two vector additions, and two matrix-vector multiplications. Taking them backwards wrt. the order that they appear, we start with the map of *recurrent_layer_activation*, which we recall to be:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{48}$$

Since the map contains no free variables, the result of the adjoint version, is simply the adjoints of elements in the array that is being mapped over.

We could attempt to performs backwards AD on the implementation of the expressions, but more simply, we can just use the known derivative of it:

$$tanh'(x) = 1 - tanh(x)^2 \tag{49}$$

We then also need to mutiply the results, with the existing adjoints for the array, which we do, and we arrive at this function for the backwards trace:

```
1 let recurrent_layer_activation_derivative y res_adj =
2   (1.0 - ((recurrent_layer_activation y) ** 2.0)) * res_adj
```

We can now look at performing backwards AD on the *vector_add* function, which looks like this:

```
1 let vector_add [n]
2                (x: [n]real) (y: [n]real) =
3   map (\(i, j) -> i + j) (zip x y)
```

As we can see, this is simply a map over 2 arrays, with the operator plus, and no free variables. We know from earlier that the derivative of a plus operation wrt. any of its inputs is simply 1. We then need to multiply with the adjoint of the result, which we should already know, hence why the backwards trace is essentially just a duplication of the adjoints of the results. But for completeness, we "compute" it like this:

```
1 let vector_add_adjoint [n]
2                -- Original inputs
3                (x: [n]real) (y: [n]real)
4                -- Adjoint of result
5                (res_adj: [n]real)
6                : ([n]real, [n]real) =
7   (map2 (\_ x_ -> 1 * x_) x res_adj, map2 (\_ y_ -> 1 * y_) y res_adj)
```

Finally, we have *matvec_mul*, which is implemented like this:

```
1 let matvec_mul [n][m] (A: [m][n]real) (B: [n]real) : [m]real =
2   map (\A_row -> reduce (+) 0 (map2 (*) A_row B)) A
```

Instead of trying to perform backwards AD on this, which is complicated since have a map with free variables in its body, we instead looked at how the operation was done, and reasoned that we could do it more simply. In the previous section, we showed an imperative program, that could find the gradients for matrix-matrix multiplication, and said that we could simply implement that, instead of a more complicated rewrite version. Implementing that in Futhark looks like this:

```
1  let matmul_adjoint [m][g][n]
2              -- Original inputs
3              (a: [m][g]real) (b: [g][n]real)
4              -- Adjoint of result
5              (res_adj: [m][n]real)
6              : ([m][g]real, [g][n]real) =
7    let a_adj = map (\i ->
8                       map (\k -> reduce (+) 0
9                                         (map2 (*) b[k,:] res_adj[i]))
10                          (iota g))
11                  (iota m)
12   let b_adj = map (\a_col ->
13                       map (\c_col -> map2 (*) c_col a_col
14                                           |> reduce (+) 0)
15                          (transpose res_adj))
16                  (transpose a)
17   in (a_adj, b_adj)
```

Here, we performed some transpositions, for performance purposes, but the result will still be the same.

With all of these steps now done, we now have a program that find the adjoint values for the weights, with respect to the loss.

8 Evaluation

The implementation of the Elman network has been validated against an implementation in PyTorch, on the following parameters:

- The implementation predicts the same values, given the same weights and inputs, both in the version that only handles a single input at a time, and the version that handles batches.
- The Backwards AD version computes the same values for the gradient, as the PyTorch implementation. The single input version was first verified against PyTorch, and then the batch version was verified against the single input version.

I have used both a small, generated data set in order to validate the gradiants, as well as a larger, real world data of stock prices[2]

[2] `https://www.kaggle.com/mattiuzc/stock-exchange-data/`

I have only implemented versions that take a single input at a time, for GRU and LSTM. I have validated that they arrive at the same output values, for a given set of inputs and weighs. I have not implemented AD on them, nor have I made a version that can handle batches of data.

Using a generated data-set and weights, we have the run times seen in figure **??**. This is for data-set with a series length of 25, 50 series, and a 20 layers deep Elman RNN. Given the nature of the Elman rnn, we would expect the run-times to grow quadraticly. We can see that, especially towards the higher input dimensionalities, it grows close to linearly. This is likely because we obtain better saturation of the GPU, since we have more parallel work to do.

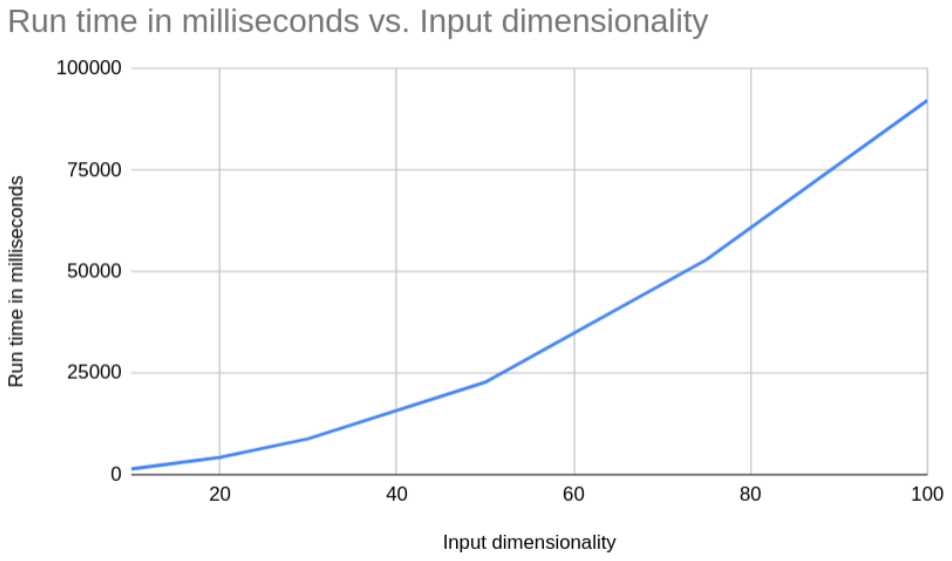

Figure 7: Run times of data-sets of varying sizes. The test was perform on a RTX 2080 Ti.

I didn't manage to run PyTorch version on the GPU, for comparison purposes, so i can't claim anything wrt. the relative performance. In some future work, one could do this, to see if we're properly taking advantage of Futhark. A parameter that you could tune on, could be "strip-mining" the loops of the code, as in, lower to number of iterations of the loops, and perform multiple iterations worth of work, in a single iteration. This would lead to more redundant work in the backwards trace, but would also lead to less memory usage. I can't say which is faster, but it would be an interesting parameter to investigate.

9 Conclusion and future work

In conclusion, we can say that it is possible to use backwards automatic differentiation, in order to calculate gradients for recurrent neural networks, in the programming language Futhark. This can be achieved by taking the original rewrite rule for backwards automatic differentiation, and creating new rules for language constructs and build in functions, in Futhark. This can be done using checkpointed intermediary values, instead of recording all intermediary values. These rules can then be used to systematically transform an implementation of a recurrent neural network, into a program that first performs a primal trace, recording necessary checkpoints, and then performs a backwards trace, that calculates the gradients of all input values, with respect to a singular output value, namely the result of the loss function. We have then validated that value of these gradients are correct, using an implementation in the machine learning framework PyTorch.

10 Acknowledgments

This projects builds on an unpublished article, titled "AD for an Array Language with Nested Parallelism", which details how automatic differentiation has been implemented as a compiler transformation in the Futhark compiler. I mainly draw inspiration from the parts, were transformation rules are explained. In a similar vein, I was also given a set of slides, which explain the transformation rules, and the mathematics behind them.

I used the "Dive into Deep Learning" book[10], for the background regarding neural networks, recurrent neural networks, activation functions and gradient descent.

I used a set of slides from Stefan Oehmcke, for the background and general understanding of recurrent neural networks, as well as the specifics of the recurrent neural networks used here.

Bibliography

[1] Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: A survey. *J. Mach. Learn. Res.*, 18(1):55955637, January 2017.

[2] KyungHyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *CoRR*, abs/1409.1259, 2014.

[3] Jeffrey L. Elman. Finding structure in time. abs/10.1207, 1990.

[4] Fabian Gieseke, Sabina Rosca, Troels Henriksen, Jan Verbesselt, and Cosmin E. Oancea. Massively-parallel change detection for satellite time series data with missing values. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 385–396, 2020.

[5] Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. LSTM: A search space odyssey. *CoRR*, abs/1503.04069, 2015.

[6] Troels Henriksen, Sune Hellfritzsch, Ponnuswamy Sadayappan, and Cosmin Oancea. Compiling generalized histograms for gpu. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[7] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. Incremental flattening for nested data parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 53–67, New York, NY, USA, 2019. ACM.

[8] Sepp Hochreiter and Jurgen Schemidhuber. Long short-term memory, 1997.

[9] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga,

and Adam Lerer. Automatic differentiation in pytorch. In *NIPS 2017 Workshop on Autodiff*, 2017.

[10] Aston Zhang, Zack C. Lipton, Mu Li, Alex J. Smola, Brent Werness, Rachel Hu, Shuai Zhang, Yi Tay, Anirudh Dagar, and Yuan Tang. Dive into deep learning, 2021.